Chapter 10. SOAP Headers

The SOAP envelope may contain a `Header` element that encompasses data outside the boundaries of an RPC (or other) style invocation. The header is an extension mechanism that can carry any kind of information that lies outside the semantics of the message in the body, but is nevertheless useful or even necessary for processing the message properly. Routing information might contain transaction identifiers, authentication keys, or other information related to the message's processing or routing.

Putting routing information in the SOAP header is useful when a message is sent to an intermediary, which in turn sends the message on to its final destination. In Chapter 9 we developed some proxy services that used another service to perform their tasks. In those cases, the client application was not aware that the service it was calling was actually a proxy. However, it's possible for proxy services to forward messages to specified endpoints or make use of other endpoints in the performance of their tasks. We'll develop an example of this kind of proxy, or intermediary, service.

Although SOAP does not currently specify any security mechanism, the `Header` element is a good place to encode authentication data such as usernames and passwords. Using the `Header` to store authentication data allows you to develop your service methods using only the parameters necessary and relevant to the method's task, and still gives you a mechanism to pass other related data to the recipient. The semantics of using the SOAP header for authentication are implementation dependent, but it's a reasonable mechanism in the absence of some other standard.

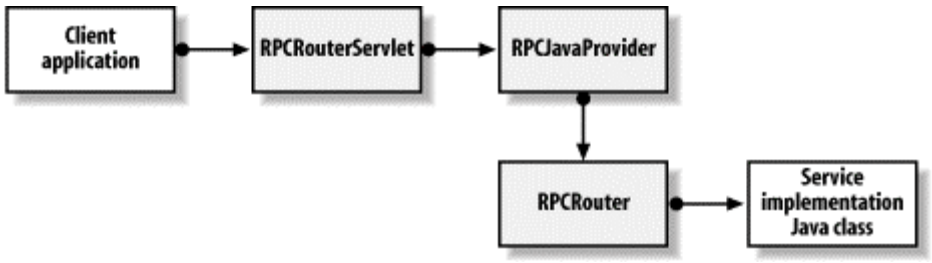
GLUE has considerable support for SOAP headers,[1] but Apache SOAP really lacks in that area. I'll walk you through a technique for working with headers in Apache SOAP client and services.

10.1 Apache SOAP Providers and Routers

The Apache SOAP engine does not provide direct support for working with SOAP headers. However, it provides just enough Java class support and functional hooks to let you add headers without too much pain. Most of the work involved will be on the service side. Basically, we want a way to gain access to the SOAP header from within the service methods we implement. But before we delve into any code, let's look at the way Apache SOAP handles and routes SOAP messages. Figure 10-1 shows the path taken by an RPC message sent to an Apache SOAP service. The objects shown in gray are part of the Apache SOAP framework, while those in white are the Java classes we've been writing throughout the book.

[1] The GLUE documentation is more than enough to get you going with headers in GLUE.

Figure 10-1. The path from the client to the service implementation



Apache SOAP is based on Java servlets. When you send an RPC-style SOAP message, you're actually sending the message to the `org.apache.soap.server.http.RPCRouterServlet` class. It's up to the servlet to determine the appropriate provider class for the specific invocation. All of our Apache SOAP services so far have used the default Java provider. Here is the relevant part of the deployment descriptor:

```
<isd:provider type="java"
```

In every case, we've assigned the `type` attribute the value `"java"`. This is nothing more than an alias for telling the framework that we want it to use the default Java provider, which is `org.apache.soap.providers.RPCJavaProvider`. The servlet uses that provider class, and all provider classes, to locate and invoke the service method. The default provider uses `org.apache.soap.server.RPCRouter` to route the message to the appropriate service implementation class. The router loads the service class and invokes the specified method using Java reflection. To specify some other provider implementation, assign the provider's full Java class name as the value of the `type` attribute.

10.2 Replacing the Provider and Router Classes

In order to make the SOAP header available to our service methods, we'll need to replace the router class with one of our own design. We could modify the `org.apache.soap.server.RPCRouter` class and rebuild the Apache SOAP source tree, but that's a bit messy; we'd have to go through the process again for every new release of the Apache SOAP engine. Luckily, Apache SOAP makes use of a concept called pluggable providers. This means that you can plug in your own provider to be used with a given service deployment. To use your own provider, you must implement your own provider class. This is just the hook we're looking for. Creating our own provider lets us use our own router, which can pass the SOAP header to our service methods. For instance, if we implement a provider class called `BetterRPCJavaProvider`, we can associate it with a service deployment by modifying the `isd:provider` element in the relevant deployment descriptor:

```
<isd:provider type="javasoap.book.ch10.services.BetterRPCJavaProvider"
```

This element tells the RPC router servlet to use our new provider class instead of the default provider. That's exactly what we want. So let's write our own provider.

All providers must implement the `org.apache.soap.util.Provider` interface, which contains `locate( )` and `invoke( )` methods. The `locate( )` method locates the service deployment information in preparation for the method invocation. The `invoke( )` method invokes the service method, assuming that a preceding call to `locate( )` was successful.

We don't need to reinvent the entire wheel here. The `org.apache.soap.providers.RPCJavaProvider` class does a perfectly good job of locating the service, so let's just extend it so that we can modify `the invoke( )` method for our purpose. Here's the code for our new provider, `BetterRPCJavaProvider`:

```
package javasoap.book.ch10.services;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.server.*;
import org.apache.soap.server.http.*;
import org.apache.soap.util.*;

public class BetterRPCJavaProvider
          extends org.apache.soap.providers.RPCJavaProvider {

   public void invoke(SOAPContext reqContext, SOAPContext resContext)
               throws SOAPException {
      try {
         Response resp = BetterRPCRouter.invoke(envelope.getHeader( ), dd,
               call, targetObject, reqContext, resContext);
         Envelope env = resp.buildEnvelope( );
         StringWriter sw = new StringWriter( );
         env.marshall(sw, call.getSOAPMappingRegistry( ), resContext);
         resContext.setRootPart(sw.toString( ),
                     Constants.HEADERVAL_CONTENT_TYPE_UTF8);
      }
      catch(Exception e) {
         if (e instanceof SOAPException)
            throw (SOAPException )e;

         throw new SOAPException(Constants.FAULT_CODE_SERVER,
                                 e.toString( ));
      }
   }
}
```

I lifted the code for the `invoke( )` method from the `org.apache.soap.providers.RPCJavaProvider` source code. The only changes I made appear in bold. The first change is to call the static `invoke( )` method of `BetterRPCRouter`, which takes an instance of `org.apache.soap.Header` as its first parameter. We haven't implemented that class yet, but we will shortly; we need it because the default router, `org.apache.soap.server.RPCRouter`, doesn't allow us to pass the header to its `invoke( )` method. The remaining parameters are the same as in `org.apache.soap.server.RPCRouter`. One benefit of extending the default provider is that it does the work of obtaining the SOAP envelope inside its implementation of the `locate( )` method, which it stores in the `envelope` class variable (an instance of `org.apache.soap.Envelope`). So all we need to do is pass `envelope.getHeader( )` as the first parameter of the `invoke( )` method of our new router class.

Now we can move on to implementing our own router. Once again, we don't need to write the whole class; instead we'll extend `org.apache.soap.server.RPCRouter`. We only need to implement the `invoke( )` method, as the rest of the base class is fine. Our `invoke( )` method

will take an `org.apache.soap.Header` as its first parameter. The trick will be finding a way to pass the header to our service class methods.

One of the nice things about having access to source code is that you can sometimes find gems that aren't otherwise noticeable. It turns out that the `org.apache.soap.server.RPCRouter` implementation actually anticipates that your service methods may be taking an instance of `org.apache.soap.rpc.SOAPContext` as their first parameter. Under those circumstances, the default router simply uses the rest of the parameters starting from position 1. You can try this if you like. Go back to any of the Apache SOAP service classes from previous chapters and stick an instance of `org.apache.soap.rpc.SOAPContext` into the method signature as its first parameter, effectively pushing the rest of the parameters to the right. If you rebuild and deploy the service after making this change, everything will work just as before; the router deals with the possibility that you included this special first parameter, but it doesn't require you to do it. This feature comes close to fulfilling our need to access the SOAP header. The problem is that obtaining the header from the `org.apache.soap.rpc.SOAPContext` is a lot of work. Still, the mechanism is perfect for our needs. The result is a router that allows us to implement service methods that can take an instance of `org.apache.soap.Header` as the first parameter. Here's the code for the `BetterRPCRouter` class. Again, most of the code comes from the base class, with the modifications shown in bold.

```
package javasoap.book.ch10.services;

import org.apache.soap.server.*;
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import org.w3c.dom.*;
import org.apache.soap.util.Bean;
import org.apache.soap.util.MethodUtils;
import org.apache.soap.util.IOUtils;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.util.StringUtils;

public class BetterRPCRouter extends org.apache.soap.server.RPCRouter {

   public static Response invoke(Header hdr, DeploymentDescriptor dd,
                     Call call, Object targetObject, SOAPContext reqCtx,
                     SOAPContext resCtx)
          throws SOAPException {
      byte providerType = dd.getProviderType(  );
      Vector params = call.getParams(  );
      String respEncStyle = call.getEncodingStyleURI(  );
      Object[] args = null;
      Class[] argTypes = null;
      if (params != null) {
         int paramsCount = params.size(  );
         args = new Object[paramsCount];
         argTypes = new Class[paramsCount];
         for (int i = 0; i < paramsCount; i++) {
            Parameter param = (Parameter)params.elementAt(i);
            args[i] = param.getValue(  );
            argTypes[i] = param.getType(  );
```

```
            if (respEncStyle == null) {
               respEncStyle = param.getEncodingStyleURI(  );
            }
         }
      }

      if (respEncStyle == null) {
         respEncStyle = Constants.NS_URI_SOAP_ENC;
      }

      Bean result = null;
      try {
         if (providerType == DeploymentDescriptor.PROVIDER_JAVA ||
             providerType == DeploymentDescriptor.PROVIDER_USER_DEFINED) {
            Method m = null ;
            try {
               m = MethodUtils.getMethod (targetObject,
                       call.getMethodName(  ), argTypes);
            } catch(NoSuchMethodException e) {
               try {
                  int paramsCount = 0 ;
                  if (params != null) paramsCount = params.size(  );
                  Class[] tmpArgTypes = new Class[paramsCount+1];
                  Object[] tmpArgs = new Object[paramsCount+1];
                  for (int i = 0 ; i < paramsCount ; i++)
                     tmpArgTypes[i+1] = argTypes[i] ;

                  argTypes = tmpArgTypes;
                  argTypes[0] = Header.class;
                  m = MethodUtils.getMethod(targetObject,
                              call.getMethodName(  ),argTypes);
                  for (int i = 0 ; i < paramsCount ; i++)
                     tmpArgs[i+1] = args[i];

                  tmpArgs[0] = hdr;
                  args = tmpArgs;
               }
               catch (NoSuchMethodException e2) {
                  throw e;
               }
            catch (Exception e2) {
               throw e2;
            }
         }          catch (Exception e) {
            throw e;
         }
         result = new Bean(m.getReturnType(  ),
                           m.invoke(targetObject, args));
      }
      else {
         Class bc = Class.forName("org.apache.soap.server.InvokeBSF");
         Class[] sig = {DeploymentDescriptor.class,
                        Object.class,
                        String.class,
                        Object[].class};
         Method m = MethodUtils.getMethod(bc, "service", sig, true);
         result = (Bean) m.invoke (null,
                    new Object[] {dd, targetObject,
                                   call.getMethodName (  ), args});
      }
   }
```

```
    catch (InvocationTargetException e) {
       Throwable t = e.getTargetException(  );
       if (t instanceof SOAPException) {
          throw (SOAPException)t;
       }
       else {
          throw new SOAPException(Constants.FAULT_CODE_SERVER,
              "Exception from service object: " + t.getMessage(  ), t);
       }
    }
    catch (ClassNotFoundException e) {
       throw new SOAPException (Constants.FAULT_CODE_SERVER,
               "Unable to load BSF: script services " +
               "unsupported without BSF", e);
    }
    catch (Throwable t) {
       throw new SOAPException (Constants.FAULT_CODE_SERVER,
         "Exception while handling service request: " +
         t.getMessage(  ),
         t);
    }

    Parameter ret = null;
    if (result.type != void.class) {
       ret = new Parameter (RPCConstants.ELEM_RETURN, result.type,
                            result.value, null);
    }

    return new Response(call.getTargetObjectURI(), call.getMethodName (  ),
                            ret, null, null, respEncStyle, resCtx);
    }
}
```

The first parameter of the `invoke( )` method is an instance of `org.apache.soap.Header` that contains the SOAP header from the request envelope. Using Java reflection, an attempt is made to obtain the service class method, assuming that the method's parameters and their associated types are a direct match to those found in the SOAP body. If that fails, the code tries again, this time looking for a method that has all of the SOAP body parameters as well as a first parameter of type `org.apache.soap.Header`. To set this up, we assign `argTypes[0]` the value `Header.class`, and assign `tmpArgs[0]` the value `hdr` (the first parameter to our `invoke( )` method). These arrays are used by the Java reflection mechanism when finding and subsequently invoking Java methods dynamically. As you can see, the vast majority of the code is unchanged. Much of it deals with finding an appropriate service class method, dealing with the possibility that a BSF script implements the service,[2] and handling faults.

10.3 An Apache SOAP Service That Handles SOAP Headers

We now have a provider and router that can pass the SOAP header to our service methods, so let's go ahead and implement a service that uses the SOAP header. The `urn:ProxyQuoteService` from Chapter 9 is a good place to start. Its implementation hardcoded the physical address of the back-end service as *http://mindstrm.com:8004/glue/urn:CorpDataServices*; we'll ask the client application to provide that information as part of the SOAP header. Our proxy service defaults to a bogus address of *http://mindstrm.com:8899/glue/urn:CorpDataServices* if no routing data appears in

[2] Apache SOAP allows you to implement services using BSF scripting. Refer to the Apache SOAP documentation if you're interested in that subject.

the header. You might use a similar technique if the service accesses a default back-end service but can also be directed to a client-specified service via the SOAP header.

Let's design our SOAP header so that it contains a single child element named `targetAddress`. This element contains the back-end service address that the proxy should use. The elements within the header may use `the SOAP-ENV:mustUnderstand` attribute with a value of `"1"` to indicate that the recipient of the envelope must understand the header element and properly use the data provided. The SOAP header sent by a client that demands the understanding of the `targetAddress` should look like this:

```
<SOAP-ENV:Header>
<targetAddress SOAP-ENV:mustUnderstand="1">
http://mindstrm.com:8004/glue/urn:CorpDataServices
</targetAddress>
</SOAP-ENV:Header>
```

So let's design our service to understand the `targetAddress` header element, but not any other header elements. If any other header elements appear with a `mustUnderstand` attribute value of `"1"`, the service returns a SOAP fault. The new service will be called `urn:DirectedQuoteProxyService`, implemented by the `DirectedQuoteProxyService` class:

```
package javasoap.book.ch10.services;

import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.util.xml.*;
import org.w3c.dom.*;

import javasoap.book.ch9.services.RemoteQuote;
import javasoap.book.ch9.services.ProxyQuote;

public class DirectedQuoteProxyService {

   static final String TARGETADDRESS = "targetAddress";
   static final String DEFAULTTARGET =
             "http://mindstrm.com:8899/glue/urn:CorpDataServices";

   public DirectedQuoteProxyService(  ) {
   }

   String getTargetAddress(Header header)
               throws SOAPException  {

      String target = DEFAULTTARGET;

      Vector entries = header.getHeaderEntries(  );
      for (int i = 0; i < entries.size(  ); i++) {
         Element el = (Element)entries.elementAt(i);
         String name = el.getLocalName(  );
         String val = el.getAttributeNS(Constants.NS_URI_SOAP_ENV,
            Constants.ATTR_MUST_UNDERSTAND);
```

```
            if (name.equals(TARGETADDRESS)) {
               target = el.getFirstChild().getNodeValue(  );
            }
            else if (val != null && val.equals("1")) {
               if (!name.equals(TARGETADDRESS))
               throw new SOAPException("MustUnderstand",
                 "Service Doesn't Understand Header Element: " + name);
            }
         }

         return target;
      }

      public ProxyQuote getQuote(Header hdr, String symbol)
                        throws SOAPException, Exception {

         String target = getTargetAddress(hdr);

         URL url = new URL(target);

         SOAPMappingRegistry smr = new SOAPMappingRegistry(  );

         BeanSerializer beanSer = new BeanSerializer(  );

         smr.mapTypes(Constants.NS_URI_SOAP_ENC,
           new QName(
            "http://www.themindelectric.com/package/" +
            "javasoap.book.ch9.services/",
            "Quote"),
            RemoteQuote.class, beanSer, beanSer);

         Call call = new Call(  );
         call.setSOAPMappingRegistry(smr);
         call.setTargetObjectURI("XYZ");
         call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

         String stock = symbol;

         Vector params = new Vector(  );
         params.addElement(new Parameter("stock", String.class, stock, null));
         call.setParams(params);

         call.setMethodName("getQuote");
         Response resp = call.invoke(url, "");
         ProxyQuote quote = new ProxyQuote(  );
         if (resp.generatedFault(  )) {
            throw new Exception("Service Call Failed");
         }
         else {
            Parameter ret = resp.getReturnValue(  );
            RemoteQuote value = (RemoteQuote)ret.getValue(  );
            quote.setStockSymbol(value.get_symbol(  ));
            quote.setLast(value.get_lastPrice(  ));
            quote.setDiff(value.get_change(  ));
            quote.setTime(value.get_timeStamp(  ));
            quote.setVol(value.get_volume(  ));
         }

         return quote;
      }
   }
```

The code is mostly the same as the `QuoteProxyClass` we developed in Chapter 9. The `getQuote( )` method now takes an `org.apache.soap.Header` as its first parameter, and a `String` containing the stock symbol as its second parameter. Before constructing the URL, a call is made to the `getTargetAddress( )` method, with the `header` passed as its parameter. The `getTargetAddress( )` method returns either the bogus default address if no `targetAddress` element appears in the header, or the value of the `targetAddress` element if it does exist. We iterate over all of the header elements, looking for the `targetAddress` element. While we're at it, we look to see if any of the other header elements contain a `SOAP-ENV:mustUnderstand` attribute with a value of `"1"`, and throw an appropriate `SOAPException` if we find any other header elements that we're required to understand. Throwing the exception results in a corresponding SOAP fault being returned to the caller. The returned value of `getTargetAddress( )` is used to construct the URL. The remainder of the code is the same as it was for `javasoap.book.ch9.services.QuoteProxyService`.

Let's deploy the service, remembering to use our `BetterRPCJavaProvider` class as the value of the `type` attribute of the `provider` element. Here's what the deployment descriptor should look like:

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
             id="urn:DirectedQuoteProxyService">
  <isd:provider type="javasoap.book.ch10.services.BetterRPCJavaProvider"
                scope="Application"
                methods="getQuote">
  <isd:java class="javasoap.book.ch10.services.DirectedQuoteProxyService"
        static="false"/>
  </isd:provider>

  <isd:faultListener>org.apache.soap.server.DOMFaultListener
  </isd:faultListener>

  <isd:mappings>
    <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         xmlns:x="urn:QuoteProxyService" qname="x:Quote"
         javaType="javasoap.book.ch9.services.ProxyQuote"
        java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
      xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
  </isd:mappings>
</isd:service>
```

Now let's write an Apache SOAP client application that can access the `urn:DirectedQuoteProxyService`. Most of the code should look familiar by now, so I'll just describe the new parts related to setting up the SOAP header. The class is called `HeaderClient`:

```
package javasoap.book.ch10.clients;

import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.util.xml.*;
import org.w3c.dom.*;
import javax.xml.parsers.DocumentBuilder;
```

```
import javasoap.book.ch9.services.ProxyQuote;

public class HeaderClient {
   public static void main(String[] args)
      throws Exception {

      URL url = new URL("http://georgetown:8080/soap/servlet/rpcrouter");

      SOAPMappingRegistry smr = new SOAPMappingRegistry(  );

      BeanSerializer beanSer = new BeanSerializer(  );

      smr.mapTypes(Constants.NS_URI_SOAP_ENC,
        new QName("urn:QuoteProxyService", "Quote"),
                  ProxyQuote.class, beanSer, beanSer);

      Call call = new Call(  );
      call.setSOAPMappingRegistry(smr);
      call.setTargetObjectURI("urn:DirectedQuoteProxyService");
      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

      String stocks = args[0];

      Header header = new Header(  );
      String must = Constants.NS_PRE_SOAP_ENV + ":" +
                                 Constants.ATTR_MUST_UNDERSTAND;

      Vector entries = new Vector(  );
      DocumentBuilder builder = XMLParserUtils.getXMLDocBuilder(  );
      Document doc = builder.newDocument(  );
      Element elem = doc.createElement("targetAddress");
      elem.setAttribute(must, "1");
      Text txt = doc.createTextNode("");
      txt.setData("http://mindstrm.com:8004/glue/urn:CorpDataServices");
      elem.appendChild(txt);
      entries.addElement(elem);
      header.setHeaderEntries(entries);
      call.setHeader(header);

      Vector params = new Vector(  );
      params.addElement(
        new Parameter("stocks", String.class, stocks, null));
      call.setParams(params);

      try {
         call.setMethodName("getQuote");
         Response resp = call.invoke(url, "");

         if (resp.generatedFault(  )) {
            Fault fault = resp.getFault(  );
            String code = fault.getFaultCode(  );
            String desc = fault.getFaultString(  );
            System.out.println(code + ": " + desc);

            Vector v = fault.getDetailEntries(  );
            int cnt = v.size(  );
            for (int i = 0; i < cnt; i++) {
               Element n = (Element)v.elementAt(i);
               Node nd = n.getFirstChild(  );
```

```
                //System.out.println(n.getNodeName(  )
                      + ": " + nd.getNodeValue(  ));
             }
          }
          else {
             Parameter ret = resp.getReturnValue(  );
             ProxyQuote value = (ProxyQuote)ret.getValue(  );
             System.out.println("Symbol:     " + value.getStockSymbol(  ) +
                       "\nLast Price: " + value.getLast(  ) +
                       "\nChange:     " + value.getDiff(  ) +
                       "\nTime Stamp: " + value.getTime(  ) +
                       "\nVolume:     " + value.getVol(  ));
          }
       }
       catch (SOAPException e) {
          System.err.println("Caught SOAPException (" +
                          e.getFaultCode(  ) + "): " +
                          e.getMessage(  ));
       }
    }
}
```

We create an instance of `org.apache.soap.Header` called `header` that we'll use to build up the header elements. The `String` variable `must` holds the qualified name of the `mustUnderstand` attribute. We build the name using the constant values `Constants.NS_PRE_SOAP_ENV` and `Constants.ATTR_MUST_UNDERSTAND`, with a ":" between them. This results in the value `SOAP-ENV:mustUnderstand`. Now we'll have to work at the XML level, like we did when we developed custom fault detail entries in Chapter 7. We take the same approach here, except that we don't need to build the `SOAP-ENV:Header` element; that's taken care of by the `org.apache.soap.Header` class. We only need to build the child entries inside the header, so we create `targetAddress` as the first element of the document. Next we set the `mustUnderstand` attribute by calling `elem.setAttribute( )`, passing it the `must` string variable as well as a value of `"1"`. Now we create a text node and set its data to `http://mindstrm.com:8004/glue/urn:CorpDataServices`, which is the endpoint we're directing the proxy service to use. The vector of header entries is added to the header by calling `header.setHeaderEntries( )`, and then the header is added to the call object using `call.setHeader( )`. Now the header we set up will be included when the SOAP envelope is created. I ran this client application with `AMX` on the command line, resulting in the following SOAP envelope being sent to the service:

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Header>
<targetAddress SOAPENV:mustUnderstand="1">
http://tokyo:8004/glue/urn:CorpDataServices
</targetAddress>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
<ns1:getQuote
xmlns:ns1="urn:DirectedQuoteProxyService"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<stocks xsi:type="xsd:string">AMX</stocks>
</ns1:getQuote>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```